1
Getting Started with SDL

Simple DirectMedia Layer (**SDL**) is a cross-platform multimedia library created by Sam Oscar Latinga. It provides low-level access to input (via mouse, keyboard, and gamepads/joysticks), 3D hardware, and the 2D video frame buffer. SDL is written in the C programming language, yet has native support for C++. The library also has bindings for several other languages such as Pascal, Objective-C, Python, Ruby, and Java; a full list of supported languages is available at `http://www.libsdl.org/languages.php`.

SDL has been used in many commercial games including World of Goo, Neverwinter Nights, and Second Life. It is also used in emulators such as ZSNES, Mupen64, and VisualBoyAdvance. Some popular games ported to Linux platforms such as Quake 4, Soldier of Fortune, and Civilization: Call to Power utilize SDL in some form.

SDL is not just used for games. It is useful for all manner of applications. If your software needs access to graphics and input, chances are that SDL will be a great help. The SDL official website has a list of applications that have been created using the library (`http://www.libsdl.org/applications.php`).

In this chapter we will cover the following:

- Getting the latest SDL build from the Mercurial repository
- Building and setting up SDL in Visual C++ 2010 Express
- Creating a window with SDL
- Implementing a basic game class

Why use SDL?

Each platform has its own way of creating and displaying windows and graphics, handling user input, and accessing any low-level hardware; each one with its own intricacies and syntax. SDL provides a uniform way of accessing these platform-specific features. This uniformity leads to more time spent tweaking your game rather than worrying about how a specific platform allows you to render or get user input, and so on. Game programming can be quite difficult, and having a library such as SDL can get your game up and running relatively quickly.

The ability to write a game on Windows and then go on to compile it on OSX or Linux with little to no changes in the code is extremely powerful and perfect for developers who want to target as many platforms as possible; SDL makes this kind of cross-platform development a breeze. While SDL is extremely effective for cross-platform development, it is also an excellent choice for creating a game with just one platform in mind, due to its ease of use and abundance of features.

SDL has a large user base and is being actively updated and maintained. There is also a responsive community along with a helpful mailing list. Documentation for SDL 2.0 is up-to-date and constantly maintained. Visiting the SDL website, `libsdl.org`, offers up lots of articles and information with links to the documentation, mailing list, and forums.

Overall, SDL offers a great place to start with game development, allowing you to focus on the game itself and ignore which platform you are developing for, until it is completely necessary. Now, with SDL 2.0 and the new features it brings to the table, SDL has become an even more capable library for game development using C++.

The best way to find out what you can do with SDL and its various functions is to use the documentation found at `http://wiki.libsdl.org/moin.cgi/CategoryAPI`. There you can see a list of all of SDL 2.0's functions along with various code examples.

What is new in SDL 2.0?

The latest version of SDL and SDL 2.0, which we will be covering in this book, is still in development. It adds many new features to the existing SDL 1.2 framework. The SDL 2.0 Roadmap (`wiki.libsdl.org/moin.cgi/Roadmap`) lists these features as:

- A 3D accelerated, texture-based rendering API
- Hardware-accelerated 2D graphics
- Support for render targets

- Multiple window support
- API support for clipboard access
- Multiple input device support
- Support for 7.1 audio
- Multiple audio device support
- Force-feedback API for joysticks
- Horizontal mouse wheel support
- Multitouch input API support
- Audio capture support
- Improvements to multithreading

While not all of these will be used in our game-programming adventures, some of them are invaluable and make SDL an even better framework to use to develop games. We will be taking advantage of the new hardware-accelerated 2D graphics to make sure our games have excellent performance.

Migrating SDL 1.2 extensions

SDL has separate extensions that can be used to add new capabilities to the library. The reason these extensions are not included in the first place is to keep SDL as lightweight as possible, with the extensions serving to add functionality only when necessary. The next table shows some useful extensions along with their purpose. These extensions have been updated from their SDL1.2/3 Versions to support SDL 2.0, and this book will cover cloning and building them from their respective repositories as and when they are needed.

Name	Description
`SDL_image`	This is an image file loading library with support for BMP, GIF, PNG, TGA, PCX, and among others.
`SDL_net`	This is a cross-platform networking library.
`SDL_mixer`	This is an audio mixer library. It has support for MP3, MIDI, and OGG.
`SDL_ttf`	This is a library supporting the use of `TrueType` fonts in SDL applications.
`SDL_rtf`	This is a library to support the rendering of the **Rich Text Format** (**RTF**).

Setting up SDL in Visual C++ Express 2010

This book will cover setting up SDL 2.0 in Microsoft's Visual C++ Express 2010 IDE. This IDE was chosen as it is available for free online, and is a widely used development environment within the games industry. The application is available at `https://www.microsoft.com/visualstudio/en-gb/express`. Once the IDE has been installed we can go ahead and download SDL 2.0. If you are not using Windows to develop games, then these instructions can be altered to suit your IDE of choice using its specific steps to link libraries and include files.

SDL 2.0 is still in development so there are no official releases as yet. The library can be retrieved in two different ways:

- One is to download the under-construction snapshot; you can then link against this to build your games (the quickest option)
- The second option is to clone the latest source using mercurial-distributed source control and build it from scratch (a good option to keep up with the latest developments of the library)

Both of these options are available at `http://www.libsdl.org/hg.php`.

Building SDL 2.0 on Windows also requires the latest DirectX SDK, which is available at `http://www.microsoft.com/en-gb/download/details.aspx?id=6812`, so make sure this is installed first.

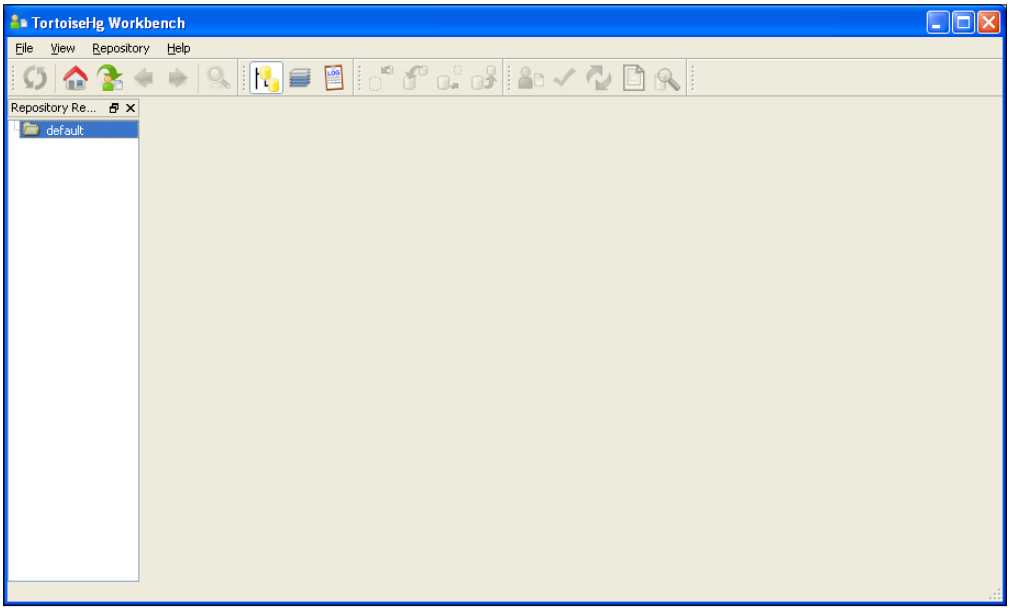
Using Mercurial to get SDL 2.0 on Windows

Getting SDL 2.0 directly from the constantly updated repository is the best way of making sure you have the latest build of SDL 2.0 and that you are taking advantage of any current bug fixes. To download and build the latest version of SDL 2.0 on Windows, we must first install a mercurial source control client so that we can mirror the latest source code and build from it. There are various command-line tools and GUIs available for use with mercurial. We will use TortoiseHg, a free and user-friendly mercurial application; it is available at `tortoisehg.bitbucket.org`. Once the application is installed, we can go ahead and grab the latest build.

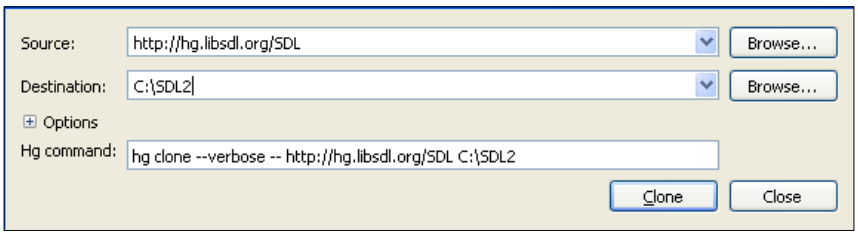
Cloning and building the latest SDL 2.0 repository

Cloning and building the latest version of SDL directly from the repository is relatively straightforward when following these steps:

1. Open up the **TortoiseHg Workbench** window.

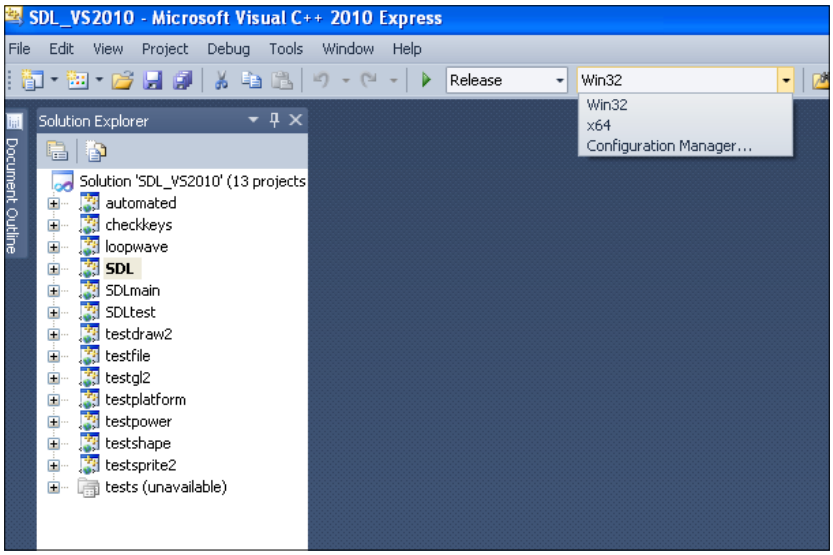


2. Pressing *Ctrl* + *Shift* + *N* will open the clone dialog box.
3. Input the source of the repository; in this case it is listed on the SDL 2.0 website as `http://hg.libsdl.org/SDL`.
4. Input or browse to choose a destination for the cloned repository—this book will assume that `C:\SDL2` is set as the location.
5. Click on **Clone** and allow the repository to copy to the chosen destination.



6. Within the `C:\SDL2` directory there will be a `VisualC` folder; inside the folder there is a Visual C++ 2010 solution, which we have to open with Visual C++ Express 2010.
7. Visual C++ Express will throw up a few errors about solution folders not being supported in the express version, but they can be safely ignored without affecting our ability to build the library.

8. Change the current build configuration to release and also choose 32 or 64 bit depending on your operating system.



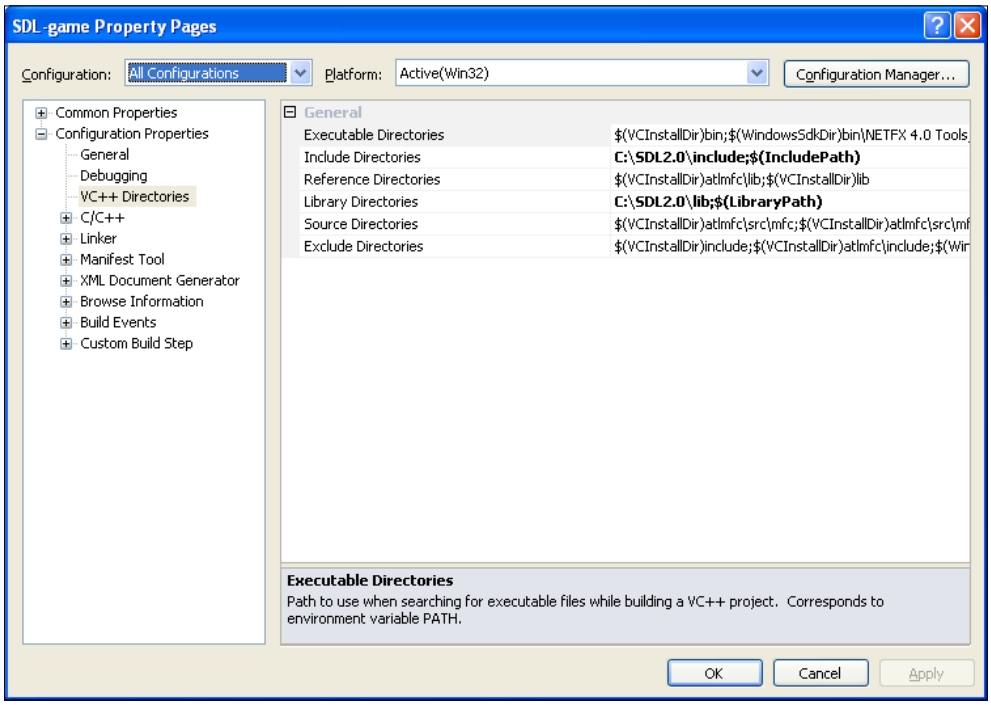
9. Right-click on the project named **SDL** listed in the **Solution Explorer** list and choose **Build**.
10. We now have a build of the SDL 2.0 library to use. It will be located at `C:\SDL2\VisualC\SDL\Win32(or x64)\Release\SDL.lib`.
11. We also need to build the SDL main library file, so choose it within the **Solution Explorer** list and build it. This file will build to `C:\SDL2\VisualC\SDLmain\Win32(or x64)\Release\SDLmain.lib`.
12. Create a folder named `lib` in `C:\SDL2` and copy `SDL.lib` and `SDLmain.lib` into this newly created folder.

I have the library; now what?

Now a Visual C++ 2010 project can be created and linked with the SDL library. Here are the steps involved:

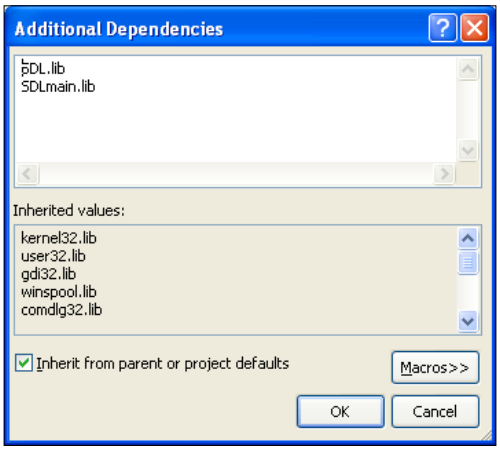
1. Create a new empty project in Visual C++ express and give it a name, such as `SDL-game`.
2. Once created, right-click on the project in the **Solution Explorer** list and choose **Properties**.

3. Change the configuration drop-down list to **All Configurations**.
4. Under **VC++ Directories**, click on **Include Directories**. A small arrow will allow a drop-down menu; click on **<Edit...>**.

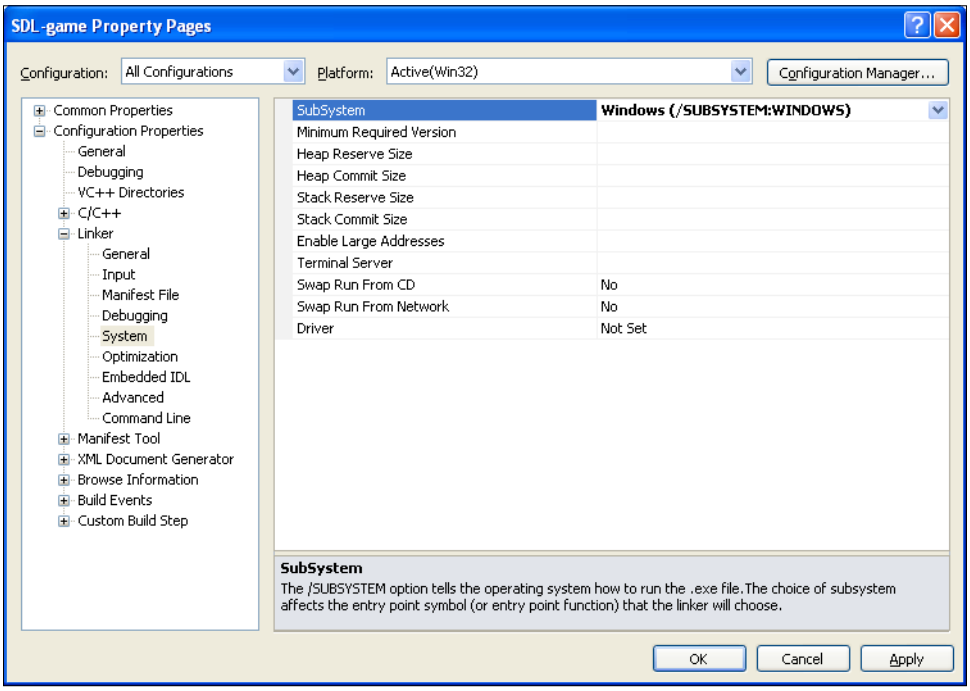


5. Double-click inside the box to create a new location. You can type or browse to `C:\SDL2.0\include` and click on **OK**.
6. Next, do the same thing under library directories, this time passing in your created `lib` folder (`C:\SDL2\lib`).

7. Next, navigate to the **Linker** heading; inside the heading there will be an **Input** choice. Inside **Additional Dependencies** type `SDL.lib SDLmain.lib`:



8. Navigate to the **System** heading and set the **SubSystem** heading to **Windows(/SUBSYSTEM:WINDOWS)**.



9. Click on **OK** and we are done.

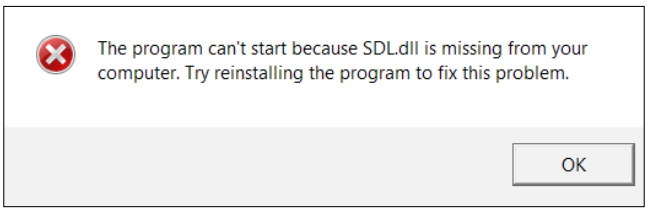
Hello SDL

We now have an empty project, which links to the SDL library, so it is time to start our SDL development. Click on **Source Files** and use the keyboard shortcut *Ctrl* + *Shift* + *A* to add a new item. Create a C++ file called `main.cpp`. After creating this file, copy the following code into the source file:

```
#include<SDL.h>

SDL_Window* g_pWindow = 0;
SDL_Renderer* g_pRenderer = 0;

int main(int argc, char* args[])
{
  // initialize SDL
  if(SDL_Init(SDL_INIT_EVERYTHING) >= 0)
  {
    // if succeeded create our window
    g_pWindow = SDL_CreateWindow("Chapter 1: Setting up SDL",
    SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
    640, 480,
    SDL_WINDOW_SHOWN);

    // if the window creation succeeded create our renderer
    if(g_pWindow != 0)
    {
      g_pRenderer = SDL_CreateRenderer(g_pWindow, -1, 0);
    }
  }
  else
  {
    return 1; // sdl could not initialize
  }

  // everything succeeded lets draw the window

  // set to black // This function expects Red, Green, Blue and
  //  Alpha as color values
  SDL_SetRenderDrawColor(g_pRenderer, 0, 0, 0, 255);

  // clear the window to black
  SDL_RenderClear(g_pRenderer);

  // show the window
```

```
    SDL_RenderPresent(g_pRenderer);

    // set a delay before quitting
    SDL_Delay(5000);

    // clean up SDL
    SDL_Quit();

    return 0;
}
```

We can now attempt to build our first SDL application. Right-click on the project and choose **Build**. There will be an error about the `SDL.dll` file not being found:



The attempted build should have created a `Debug` or `Release` folder within the project directory (usually located in your `Documents` folder under visual studio and projects). This folder contains the `.exe` file from our attempted build; we need to add the `SDL.dll` file to this folder. The `SDL.dll` file is located at `C:\SDL2\VisualC\SDL\Win32` (or `x64)\Release\SDL.dll l`). When you want to distribute your game to another computer, you will have to share this file as well as the executable. After you have added the `SDL.dll` file to the executable folder, the project will now compile and show an SDL window; wait for 5 seconds and then close.

An overview of Hello SDL

Let's go through the `Hello SDL` code:

1. First, we included the `SDL.h` header file so that we have access to all of SDL's functions:

   ```
   #include<SDL.h>
   ```

2. The next step is to create some global variables. One is a pointer to an `SDL_Window` function, which will be set using the `SDL_CreateWindow` function. The second is a pointer to an `SDL_Renderer` object; set using the `SDL_CreateRenderer` function:

   ```
   SDL_Window* g_pWindow = 0;
   SDL_Renderer* g_pRenderer = 0;
   ```

3. We can now initialize SDL. This example initializes all of SDL's subsystems using the `SDL_INIT_EVERYTHING` flag, but this does not always have to be the case (see SDL initialization flags):

```
int main(int argc, char* argv[])
{
  // initialize SDL
  if(SDL_Init(SDL_INIT_EVERYTHING) >= 0)
   {
```

4. If the SDL initialization was successful, we can create the pointer to our window. `SDL_CreateWindow` returns a pointer to a window matching the passed parameters. The parameters are the window title, *x* position of the window, *y* position of the window, width, height, and any required `SDL_flags` (we will cover these later in the chapter). `SDL_WINDOWPOS_CENTERED` will center our window relative to the screen:

```
// if succeeded create our window
g_pWindow = SDL_CreateWindow("Chapter 1: Setting up SDL", SDL_
WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 640, 480, SDL_WINDOW_
SHOWN);
```

5. We can now check whether the window creation was successful, and if so, move on to set the pointer to our renderer, passing the window we want the renderer to use as a parameter; in our case, it is the newly created `g_pWindow` pointer. The second parameter passed is the index of the rendering driver to initialize; in this case, we use `-1` to use the first capable driver. The final parameter is `SDL_RendererFlag` (see SDL renderer flags):

```
// if the window creation succeeded create our renderer
if(g_pWindow != 0)
{
  g_pRenderer = SDL_CreateRenderer(g_pWindow, -1, 0);
}
else
{
  return 1; // sdl could not initialize
}
```

6. If everything was successful, we can now create and show our window:

```
// everything succeeded lets draw the window

  // set to black
SDL_SetRenderDrawColor(g_pRenderer, 0, 0, 0, 255);
```

```
   // clear the window to black
SDL_RenderClear(g_pRenderer);

   // show the window
SDL_RenderPresent(g_pRenderer);

   // set a delay before quitting
SDL_Delay(5000);

   // clean up SDL
SDL_Quit();
```

SDL initialization flags

Event handling, file I/O, and threading subsystems are all initialized by default in SDL. Other subsystems can be initialized using the following flags:

Flag	Initialized subsystem(s)
`SDL_INIT_HAPTIC`	Force feedback subsystem
`SDL_INIT_AUDIO`	Audio subsystem
`SDL_INIT_VIDEO`	Video subsystem
`SDL_INIT_TIMER`	Timer subsystem
`SDL_INIT_JOYSTICK`	Joystick subsystem
`SDL_INIT_EVERYTHING`	All subsystems
`SDL_INIT_NOPARACHUTE`	Don't catch fatal signals

We can also use bitwise (`|`) to initialize more than one subsystem. To initialize only the audio and video subsystems, we can use a call to `SDL_Init`, for example:

```
SDL_Init(SDL_INIT_AUDIO | SDL_INIT_VIDEO);
```

Checking whether a subsystem has been initialized or not can be done with a call to the `SDL_WasInit()` function:

```
if(SDL_WasInit(SDL_INIT_VIDEO) != 0)
{
  cout << "video was initialized";
}
```

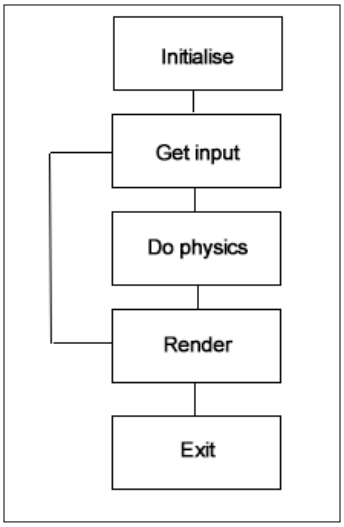
SDL renderer flags

When initializing an `SDL_Renderer` flag, we can pass in a flag to determine its behavior. The following table describes each flag's purpose:

Flag	Purpose
`SDL_RENDERER_SOFTWARE`	Use software rendering
`SDL_RENDERER_ACCELERATED`	Use hardware acceleration
`SDL_RENDERER_PRESENTVSYNC`	Synchronize renderer update with screen's refresh rate
`SDL_RENDERER_TARGETTEXTURE`	Supports render to texture

What makes up a game

Outside the design and gameplay of a game, the underlying mechanics are essentially the interaction of various subsystems such as graphics, game logic, and user input. The graphics subsystem should not know how the game logic is implemented or vice versa. We can think of the structure of a game as follows:



Once the game is initialized, it then goes into a loop of checking for user input, updating any values based on the game physics, before rendering to the screen. Once the user chooses to exit, the loop is broken and the game moves onto cleaning everything up and exiting. This is the basic scaffold for a game and it is what will be used in this book.

We will be building a reusable framework that will take all of the legwork out of creating a game in SDL 2.0. When it comes to boilerplate code and setup code, we really only want to write it once and then reuse it within new projects. The same can be done with drawing code, event handling, map loading, game states, and anything else that all games may require. We will start by breaking up the Hello SDL 2.0 example into separate parts. This will help us to start thinking about how code can be broken into reusable standalone chunks rather than packing everything into one large file.

Breaking up the Hello SDL code

We can break up the Hello SDL into separate functions:

```
bool g_bRunning = false; // this will create a loop
```

Follow these steps to break the `Hello SDL` code:

1. Create an `init` function after the two global variables that takes any necessary values as parameters and passes them to the `SDL_CreateWindow` function:

```
bool init(const char* title, int xpos, int ypos, int
height, int width, int flags)
{
  // initialize SDL
  if(SDL_Init(SDL_INIT_EVERYTHING) >= 0)
  {
    // if succeeded create our window
    g_pWindow = SDL_CreateWindow(title, xpos, ypos,
    height, width, flags);

    // if the window creation succeeded create our
    renderer
    if(g_pWindow != 0)
    {
      g_pRenderer = SDL_CreateRenderer(g_pWindow, -1, 0);
    }
  }
  else
  {
    return false; // sdl could not initialize
  }

  return true;
```

```
}

void render()
{
  // set to black
  SDL_SetRenderDrawColor(g_pRenderer, 0, 0, 0, 255);

  // clear the window to black
  SDL_RenderClear(g_pRenderer);

  // show the window
  SDL_RenderPresent(g_pRenderer);
}
```

2. Our main function can now use these functions to initialize SDL:

```
int main(int argc, char* argv[])
{
  if(init("Chapter 1: Setting up SDL",
  SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 640,
  480, SDL_WINDOW_SHOWN))
  {
    g_bRunning = true;
  }
  else
  {
    return 1; // something's wrong
  }

  while(g_bRunning)
  {
    render();
  }

  // clean up SDL
  SDL_Quit();

  return 0;
}
```

As you can see, we have broken the code up into separate parts: one function does the initialization for us and the other does the rendering code. We've added a way to keep the program running in the form of a `while` loop that runs continuously, rendering our window.

Let's take it a step further and try to identify which separate parts a full game might have and how our main loop might look. Referring to the first screenshot, we can see that the functions we will need are `initialize`, `get input`, `do physics`, `render`, and `exit`. We will generalize these functions slightly and rename them to `init()`, `handleEvents()`, `update()`, `render()`, and `clean()`. Let's put these functions into `main.cpp`:

```
void init(){}
void render(){}
void update(){}
void handleEvents(){}
void clean(){}

bool g_bRunning = true;

int main()
{
  init();

  while(g_bRunning)
  {
    handleEvents();
    update();
    render();
  }

  clean();
}
```

What does this code do?

This code does not do much at the moment, but it shows the bare bones of a game and how a main loop might be broken apart. We declare some functions that can be used to run our game: first, the `init()` function, which will initialize SDL and create our window, and second, we declare the core loop functions of `render`, `update`, and `handle events`. We also declare a `clean` function, which will clean up code at the end of our game. We want this loop to continue running so we have a Boolean value that is set to `true`, so that we can continuously call our core loop functions.

The Game class

So, now that we have an idea of what makes up a game, we can separate the functions into their own class by following these steps:

1. Go ahead and create a new file in the project called `Game.h`:

```
#ifndef __Game__
#define __Game__

class Game
{
};

#endif /* defined(__Game__) */
```

2. Next, we can move our functions from the `main.cpp` file into the `Game.h` header file:

```
class Game
{
public:

  Game() {}
  ~Game() {}

  // simply set the running variable to true
  void init() { m_bRunning = true; }

  void render(){}
  void update(){}
  void handleEvents(){}
  void clean(){}

  // a function to access the private running variable
  bool running() { return m_bRunning; }

private:

  bool m_bRunning;
};
```

3. Now, we can alter the `main.cpp` file to use this new `Game` class:

```
#include "Game.h"

// our Game object
Game* g_game = 0;

int main(int argc, char* argv[])
{
  g_game = new Game();

  g_game->init("Chapter 1", 100, 100, 640, 480, 0);

  while(g_game->running())
  {
    g_game->handleEvents();
    g_game->update();
    g_game->render();
  }
  g_game->clean();

  return 0;
}
```

   Our `main.cpp` file now does not declare or define any of these functions; it simply creates an instance of `Game` and calls the needed methods.

4. Now that we have this skeleton code, we can go ahead and tie SDL into it to create a window; we will also add a small event handler so that we can exit the application rather than having to force it to quit. We will slightly alter our `Game.h` file to allow us to add some SDL specifics and to also allow us to use an implementation file instead of defining functions in the header:

```
#include "SDL.h"

class Game
{
public:

  Game();
  ~Game();

  void init();

  void render();
```

```
    void update();
    void handleEvents();
    void clean();

    bool running() { return m_bRunning; }

  private:

    SDL_Window* m_pWindow;
    SDL_Renderer* m_pRenderer;

    bool m_bRunning;
  };
```

Looking back at the first part of this chapter (where we created an SDL window), we know that we need a pointer to an `SDL_Window` object that is set when calling `SDL_CreateWindow`, and a pointer to an `SDL_Renderer` object that is created by passing our window into `SDL_CreateRenderer`. The `init` function can be extended to use the same parameters as in the initial sample as well. This function will now return a Boolean value so that we can check whether SDL is initialized correctly:

```
bool init(const char* title, int xpos, int ypos, int width, int
height, int flags);
```

We can now create a new implementation `Game.cpp` file in the project so that we can create the definitions for these functions. We can take the code from the *Hello SDL* section and add it to the functions in our new `Game` class.

Open up `Game.cpp` and we can begin adding some functionality:

1. First, we must include our `Game.h` header file:

   ```
   #include "Game.h"
   ```

2. Next, we can define our `init` function; it is essentially the same as the `init` function we have previously written in our `main.cpp` file:

   ```
   bool Game::init(const char* title, int xpos, int ypos, int width,
   int height, int flags)
   {
     // attempt to initialize SDL
     if(SDL_Init(SDL_INIT_EVERYTHING) == 0)
     {
       std::cout << "SDL init success\n";
       // init the window
       m_pWindow = SDL_CreateWindow(title, xpos, ypos,
       width, height, flags);
   ```

```
    if(m_pWindow != 0) // window init success
    {
      std::cout << "window creation success\n";
      m_pRenderer = SDL_CreateRenderer(m_pWindow, -1, 0);

      if(m_pRenderer != 0) // renderer init success
      {
        std::cout << "renderer creation success\n";
        SDL_SetRenderDrawColor(m_pRenderer,
        255,255,255,255);
      }
      else
      {
        std::cout << "renderer init fail\n";
        return false; // renderer init fail
      }
    }
    else
    {
      std::cout << "window init fail\n";
      return false; // window init fail
    }
  }
  else
  {
    std::cout << "SDL init fail\n";
    return false; // SDL init fail
  }

  std::cout << "init success\n";
  m_bRunning = true; // everything inited successfully,
  start the main loop

  return true;
}
```

3. We will also define the `render` function. It clears the renderer and then renders again with the clear color:

```
void Game::render()
{
  SDL_RenderClear(m_pRenderer); // clear the renderer to
  the draw color

  SDL_RenderPresent(m_pRenderer); // draw to the screen
}
```

4. Finally, we can clean up. We destroy both the window and the renderer and also call the `SDL_Quit` function to close all the subsystems:

```
{
  std::cout << "cleaning game\n";
  SDL_DestroyWindow(m_pWindow);
  SDL_DestroyRenderer(m_pRenderer);
  SDL_Quit();
}
```

So we have moved the `Hello SDL 2.0` code from the `main.cpp` file into a class called `Game`. We have freed up the `main.cpp` file to handle only the `Game` class; it knows nothing about SDL or how the `Game` class is implemented. Let's add one more thing to the class to allow us to close the application the regular way:

```
void Game::handleEvents()
{
  SDL_Event event;
  if(SDL_PollEvent(&event))
  {
    switch (event.type)
    {
      case SDL_QUIT:
        m_bRunning = false;
      break;

      default:
      break;
    }
  }
}
```

We will cover event handling in more detail in a forthcoming chapter. What this function now does is check if there is an event to handle, and if so, check if it is an `SDL_QUIT` event (by clicking on the cross to close a window). If the event is `SDL_QUIT`, we set the `Game` class' `m_bRunning` member variable to `false`. The act of setting this variable to `false` makes the main loop stop and the application move onto cleaning up and then exiting:

```
void Game::clean()
{
  std::cout << "cleaning game\n";
  SDL_DestroyWindow(m_pWindow);
  SDL_DestroyRenderer(m_pRenderer);
  SDL_Quit();
}
```

The `clean()` function destroys the window and renderer and then calls the `SDL_Quit()` function, closing all the initialized SDL subsystems.

To enable us to view our `std::cout` messages, we must first include `Windows.h` and then call `AllocConsole(); andfreopen("CON", "w", stdout);`. You can do this in the `main.cpp` file. Just remember to remove it when sharing your game.

Fullscreen SDL

`SDL_CreateWindow` takes an enumeration value of type `SDL_WindowFlags`. These values set how the window will behave. We created an `init` function in our `Game` class:

```
bool init(const char* title, int xpos, int ypos, int width, int
height, int flags);
```

The final parameter is an `SDL_WindowFlags` value, which is then passed into the `SDL_CreateWindow` function when initializing:

```
// init the window
m_pWindow = SDL_CreateWindow(title, xpos, ypos, width, height, flags);
```

Here is a table of the `SDL_WindowFlags` function:

Flag	Purpose
`SDL_WINDOW_FULLSCREEN`	Make the window fullscreen
`SDL_WINDOW_OPENGL`	Window can be used with as an OpenGL context
`SDL_WINDOW_SHOWN`	The window is visible
`SDL_WINDOW_HIDDEN`	Hide the window
`SDL_WINDOW_BORDERLESS`	No border on the window
`SDL_WINDOW_RESIZABLE`	Enable resizing of the window
`SDL_WINDOW_MINIMIZED`	Minimize the window
`SDL_WINDOW_MAXIMIZED`	Maximize the window
`SDL_WINDOW_INPUT_GRABBED`	Window has grabbed input focus
`SDL_WINDOW_INPUT_FOCUS`	Window has input focus
`SDL_WINDOW_MOUSE_FOCUS`	Window has mouse focus
`SDL_WINDOW_FOREIGN`	The window was not created using SDL

Let's pass in `SDL_WINDOW_FULLSCREEN` to the `init` function and test out some fullscreen SDL. Open up the `main.cpp` file and add this flag:

```
g_game->init("Chapter 1", 100, 100, 640, 580, SDL_WINDOW_FULLSCREEN))
```

Build the application again and you should see that the window is fullscreen. To exit the application, it will have to be forced to quit (*Alt* + *F4* on Windows); we will be able to use the keyboard to quit the application in forthcoming chapters, but for now, we won't need fullscreen. One problem we have here is that we have now added something SDL specific to the `main.cpp` file. While we will not use any other frameworks in this book, in future we may want to use another. We can remove this SDL-specific flag and replace it with a Boolean value for whether we want fullscreen or not.

Replace the `int flags` parameter in our `Game init` function with a `boolfullscreen` parameter:

- The code snippet for `Game.h`:

```
bool init(const char* title, int xpos, int ypos, int width, int
height, bool fullscreen);
```

- The code snippet for `Game.cpp`:

```
bool Game::init(const char* title, int xpos, int ypos, int width,
int height, bool fullscreen)
{
  int flags = 0;

  if(fullscreen)
  {
    flags = SDL_WINDOW_FULLSCREEN;
  }
}
```

We create an `int` flags variable to pass into the `SDL_CreateWindow` function; if we have set `fullscreen` to `true`, then this value will be set to the `SDL_WINDOW_FULLSCREEN` flag, otherwise it will remain as `0` to signify that no flags are being used. Let's test this now in our `main.cpp` file:

```
if(g_game->init("Chapter 1", 100, 100, 640, 480, true))
```

This will again set our window to fullscreen, but we aren't using the SDL-specific flag to do it. Set it to `false` again as we will not need fullscreen for a while. Feel free to try out a few of the other flags to see what effects they have.

Summary

A lot of ground has been covered in this chapter. We learned what SDL is and why it is a great tool for game development. We looked at the overall structure of a game and how it can be broken into individual parts, and we started to develop the skeleton of our framework by creating a `Game` class that can be used to initialize SDL and render things to the screen. We also had a small look at how SDL handles events by listening for a `quit` event to close our application. In the next chapter we will look at drawing in SDL and building the `SDL_image` extension.

2
Drawing in SDL

Graphics are very important to games and they can also be one of the main performance bottlenecks if not handled correctly. With SDL 2.0 we can really take advantage of the GPU when rendering, which gives us a real boost in terms of the speed of rendering.

In this chapter we will cover:

- The basics of drawing with SDL
- Source and destination rectangles
- Loading and displaying textures
- Using the `SDL_image` extension

Basic SDL drawing

In the previous chapter we created an SDL window but we have yet to render anything to the screen. SDL can use two structures to draw to the screen. One is the `SDL_Surface` structure, which contains a collection of pixels and is rendered using software rendering processes (not the GPU). The other is `SDL_Texture`; this can be used for hardware-accelerated rendering. We want our games to be as efficient as possible so we will focus on using `SDL_Texture`.

Getting some images

We need some images to load throughout this chapter. We do not want to spend any time creating art assets for our games at this point; we want to focus entirely on the programming side. In this book we will use assets from the `SpriteLib` collection available at `http://www.widgetworx.com/widgetworx/portfolio/spritelib.html`.

I have altered some of these files to allow us to easily use them in the upcoming chapters. These images are available with the source code download for this book. The first one we will use is the `rider.bmp` image file:

Creating an SDL texture

First we will create a pointer to an `SDL_Texture` object as a member variable in our `Game.h` header file. We will also create some rectangles to be used when drawing the texture.

```
SDL_Window* m_pWindow;
SDL_Renderer* m_pRenderer;

SDL_Texture* m_pTexture; // the new SDL_Texture variable
SDL_Rect m_sourceRectangle; // the first rectangle
SDL_Rect m_destinationRectangle; // another rectangle
```

We can load this texture in our game's `init` function for now. Open up `Game.cpp` and follow the steps to load and draw an `SDL_Texture`:

1. First we will make an assets folder to hold our images, place this in the same folder as your source code (not the executable code). When you want to distribute the game you will copy this assets folder along with your executable. But for development purposes we will keep it in the same folder as the source code. Place the `rider.bmp` file into this assets folder.
2. In our game's `init` function we can load our image. We will use the `SDL_LoadBMP` function which returns an `SDL_Surface*`. From this `SDL_Surface*` we can create `SDL_Texture` structure using the `SDL_CreateTextureFromSurface` function. We then free the temporary surface, releasing any used memory.

   ```
   SDL_Surface* pTempSurface = SDL_LoadBMP("assets/rider.bmp");

   m_pTexture = SDL_CreateTextureFromSurface(m_pRenderer,
   pTempSurface);

   SDL_FreeSurface(pTempSurface);
   ```

3. We now have `SDL_Texture` ready to be drawn to the screen. We will first get the dimensions of the texture we have just loaded, and use them to set the width and height of `m_sourceRectangle` so that we can draw it correctly.

```
SDL_QueryTexture(m_pTexture, NULL, NULL,
&m_sourceRectangle.w, &m_sourceRectangle.h);
```

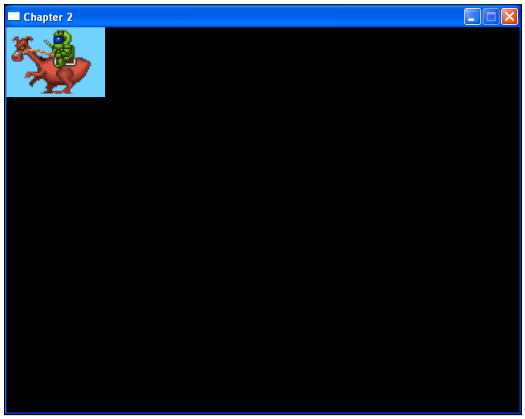
4. Querying the texture will allow us to set the width and height of our source rectangle to the exact dimensions needed. So now that we have the correct height and width of our texture stored in `m_sourceRectangle` we must also set the destination rectangle's height and width. This is done so that our renderer knows which part of the window to draw our image to, and also the width and height of the image we want to render. We will set both x and y coordinates to `0` (top left). Window coordinates can be represented with an `x` and `y` value, with `x` being the horizontal position and `y` the vertical. Therefore the coordinates for the top-left of a window in SDL would be (0,0) and the center point would be the width of the window divided by two for `x`, and the height of the window divided by two for `y`.

```
m_destinationRectangle.x = m_sourceRectangle.x = 0;
m_destinationRectangle.y = m_sourceRectangle.y = 0;
m_destinationRectangle.w = m_sourceRectangle.w;
m_destinationRectangle.h = m_sourceRectangle.h;
```

5. Now that we have a loaded texture and its dimensions, we can move on to rendering it to the screen. Move to our game's `render` function and we will add the code to draw our texture. Put this function between the calls to `SDL_RenderClear` and `SDL_RenderPresent`.

```
SDL_RenderCopy(m_pRenderer, m_pTexture, &m_sourceRectangle,
&m_destinationRectangle);
```

6. Build the project and you will see our loaded texture.

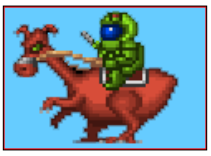


Source and destination rectangles

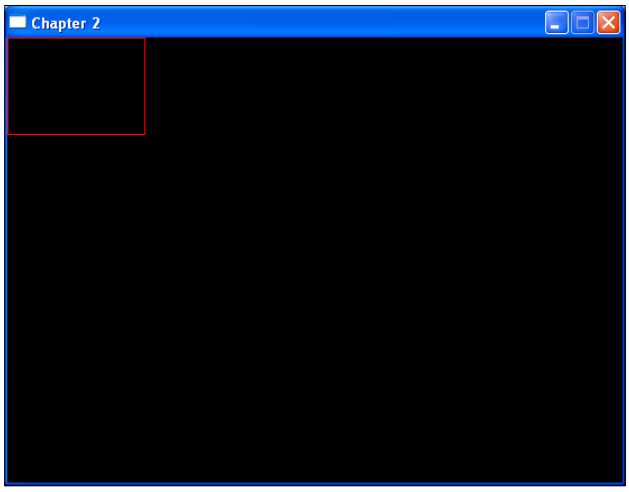
Now that we have something drawn to the screen, it is a good idea to cover the purpose of source and destination rectangles, as they will be extremely important for topics such as tile map loading and drawing. They are also important for sprite sheet animation which we will be covering later in this chapter.

We can think of a source rectangle as defining the area we want to copy from a texture onto the window:

1. In the previous example, we used the entire image so we could simply define the source rectangle's dimensions with the same dimensions as those of the loaded texture.



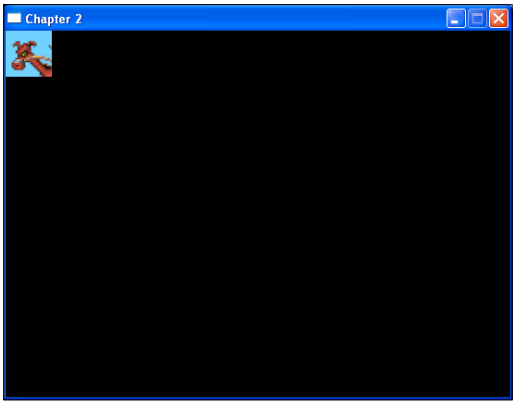
2. The red box in the preceding screenshot is a visual representation of the source rectangle we used when drawing to the screen. We want to copy pixels from inside the source rectangle to a specific area of the renderer, the destination rectangle (the red box in the following screenshot).



3. As you would expect, these rectangles can be defined however you wish. For example, let's open up our `Game.cpp` file again and take a look at changing the size of the source rectangle. Place this code after the `SDL_QueryTexture` function.

```
m_sourceRectangle.w = 50;
m_sourceRectangle.h = 50;
```

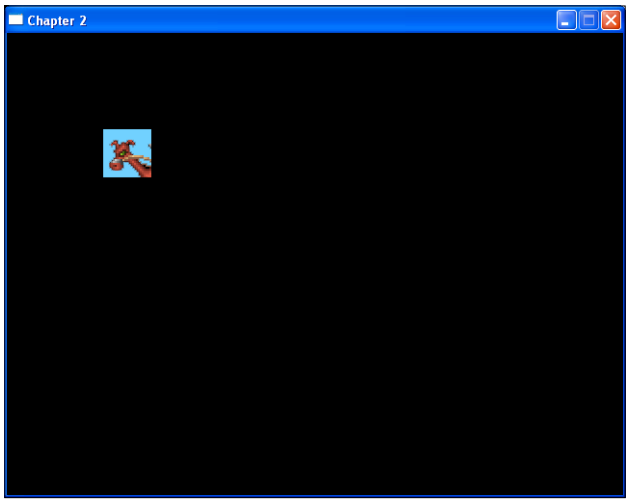
Now build again and you should see that only a 50 x 50 square of the image has been copied across to the renderer.



4. Now let us move the destination rectangle by changing its `x` and `y` values.

```
m_destinationRectangle.x = 100;
m_destinationRectangle.y = 100;
```

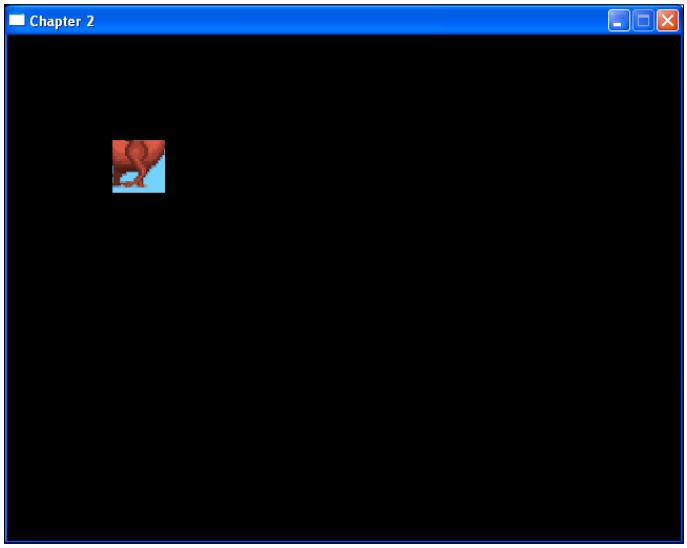
Build the project again and you will see that our source rectangle location has remained the same but the destination rectangle has moved. All we have done is move the location that we want the pixels inside the source rectangle to be copied to.



5. So far we have left the source rectangle's `x` and `y` coordinates at 0 but they can also be moved around to only draw the section of the image that you want. We can move the `x` and `y` coordinates of the source to draw the bottom-right section of the image rather than the top-left. Place this code just before where we set the destination rectangle's location.

```
m_sourceRectangle.x = 50;
m_sourceRectangle.y = 50;
```

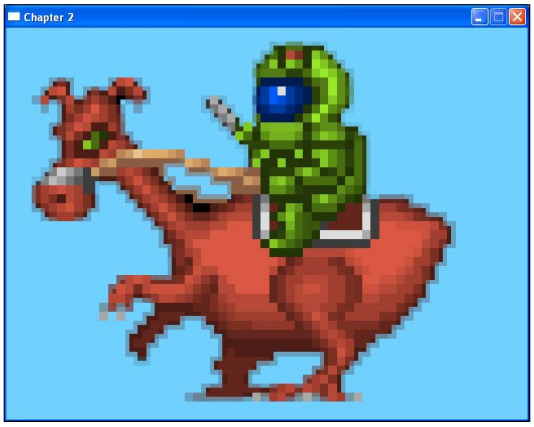
   You can see that we are still drawing to the same destination location but we are copying a different 50 x 50 section of the image.



6. We can also pass null into the render copy for either rectangle.

```
SDL_RenderCopy(m_pRenderer, m_pTexture, 0, 0);
```

   Passing null into the source rectangle parameter will make the renderer use the entire texture. Likewise, passing null to the destination rectangle parameter will use the entire renderer for display.



We have covered a few different ways that we can use rectangles to define areas of images that we would like to draw. We will now put that knowledge into practice by displaying an animated sprite sheet.

Animating a sprite sheet

We can apply our understanding of source and destination rectangles to the animation of a sprite sheet. A sprite sheet is a series of animation frames all put together into one image. The separate frames need to have a very specific width and height so that they create a seamless motion. If one part of the sprite sheet is not correct it will make the whole animation look out of place or completely wrong. Here is an example sprite sheet that we will use for this demonstration:

1. This animation is six frames long and each frame is 128 x 82 pixels. We know from the previous section that we can use a source rectangle to grab a certain part of an image. Therefore we can start by defining a source rectangle that encompasses the first frame of the animation only.

2. Since we know the width, height, and location of the frame on the sprite sheet we can go ahead and hardcode these values into our source rectangle. First we must load the new `animate.bmp` file. Place it into your assets folder and alter the loading code.

```
SDL_Surface* pTempSurface =
SDL_LoadBMP("assets/animate.bmp");
```

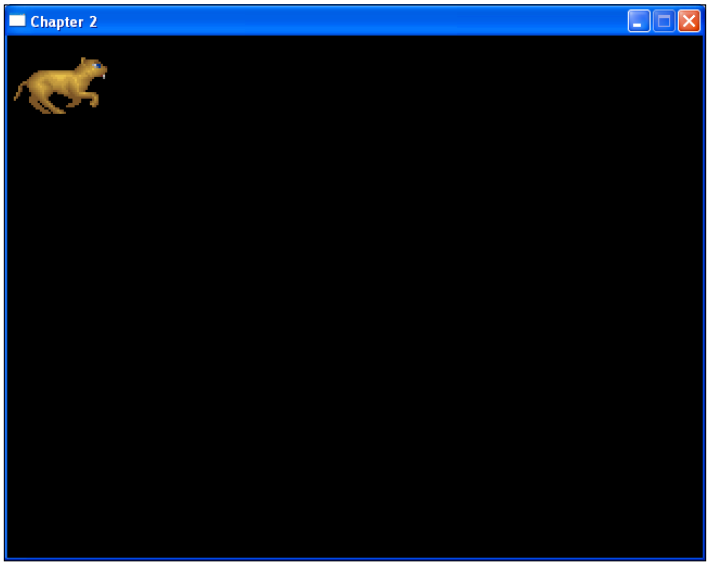
3. This will now load our new sprite sheet BMP. We can remove the `SDL_QueryTexture` function as we are now defining our own sizes. Alter the size of the source rectangle to only get the first frame of the sheet.

```
m_sourceRectangle.w = 128;
m_sourceRectangle.h = 82;
```

4. We will leave the `x` and `y` position of both rectangles at `0` so that we draw the image from the top-left corner and also copy it to the top-left corner of the renderer. We will also leave the dimensions of the destination rectangle as we want it to remain the same as the source rectangle. Pass both rectangles into the `SDL_RenderCopy` function:

```
SDL_RenderCopy(m_pRenderer, m_pTexture, &m_sourceRectangle,
&m_destinationRectangle);
```

   Now when we build we will have the first frame of the animation.



5. Now that we have the first frame, we can move on to animating the sprite sheet. Each frame has the exact same dimensions. This is extremely important for this sheet to animate correctly. All we want to do is move the location of the source rectangle, not its dimensions.

6. Every time we want to move another frame, we simply move the location of the source rectangle and copy it to the renderer. To do this we will use our `update` function.

```
void Game::update()
{
  m_sourceRectangle.x = 128 * int(((SDL_GetTicks() / 100) % 6));
}
```

7. Here we have used `SDL_GetTicks()` to find out the amount of milliseconds since SDL was initialized. We then divide this by the amount of time (in ms) we want between frames and then use the modulo operator to keep it in range of the amount of frames we have in our animation. This code will (every 100 milliseconds) shift the `x` value of our source rectangle by 128 pixels (the width of a frame), multiplied by the current frame we want, giving us the correct position. Build the project and you should see the animation displayed.

Flipping images

In most games, players, enemies, and so on, will move in more than one direction. To allow the sprite to face in the direction it is moving we will have to flip our sprite sheet. We could of course create a new row in our sprite sheet with the frames flipped, but this would use more memory, which we do not want. SDL 2.0 has another render function that allows us to pass in the way we want our image to be flipped or rotated. The function we will use is `SDL_RenderCopyEx`. This function takes the same parameters as `SDL_RenderCopy` but also takes specific parameters for rotation and flipping. The fourth parameter is the angle we want the image to be displayed with parameter five being the center point we want for the rotation. The final parameter is an enumerated type called `SDL_RendererFlip`.

The following table shows the available values for the `SDL_RendererFlip` enumerated type:

SDL_RendererFlip value	Purpose
`SDL_FLIP_NONE`	No flipping
`SDL_FLIP_HORIZONTAL`	Flip the texture horizontally
`SDL_FLIP_VERTICAL`	Flip the texture vertically

We can use this parameter to flip our image. Here is the revised render function:

```
void Game::render()
{
  SDL_RenderClear(m_pRenderer);

  SDL_RenderCopyEx(m_pRenderer, m_pTexture,
  &m_sourceRectangle, &m_destinationRectangle,
  0, 0, SDL_FLIP_HORIZONTAL); // pass in the horizontal flip

  SDL_RenderPresent(m_pRenderer);
}
```

Build the project and you will see that the image has been flipped and is now facing to the left. Our characters and enemies will also have frames specifically for animations such as attack and jump. These can be added to different rows of the sprite sheet and the source rectangle's `y` value is incremented accordingly. (We will cover this in more detail when we create our game objects.)

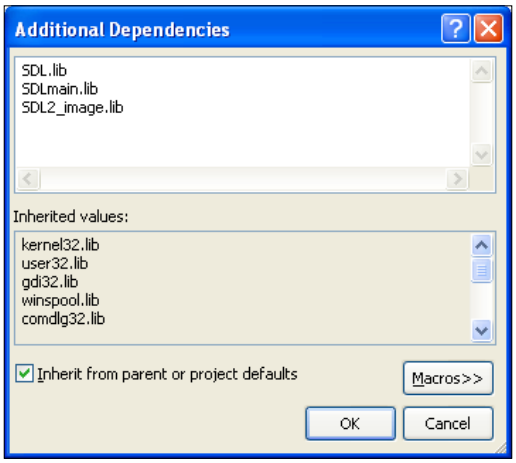
Installing SDL_image

So far we have only been loading BMP image files. This is all that SDL supports without any extensions. We can use `SDL_image` to enable us to load many different image file types such as BMP, GIF, JPEG, LBM, PCX, PNG, PNM, TGA, TIFF, WEBP, XCF, XPM, and XV. First we will need to clone the latest build of `SDL_image` to ensure it will work with SDL 2.0:

1. Open up the `TortoiseHg` workbench and use *Ctrl* + *Shift* + *N* to clone a new repository.
2. The repository for SDL_image is listed on `http://www.libsdl.org/projects/SDL_image/` and `http://hg.libsdl.org/SDL_image/`. So let's go ahead and type that into the **Source** box.

3. Our destination will be a new directory, `C:\SDL2_image`. After typing this into the **Destination** box, hit **clone** and wait for it to complete.
4. Once you have created this folder, navigate to our `C:\SDL2_image` cloned repository. Open up the `VisualC` folder and then open the `SDL_image_VS2010` VC++ project with Visual Studio 2010 express.
5. Right-click on the `SDL2_image` project and then click on **Properties**. Here we have to include the `SDL.h` header file. Change the configuration to **All Configurations**, navigate to **VC++ Directories**, click on the **Include Directories** drop-down, and then on **<Edit…>**. Here we can put in our `C:\SDL2\include\` directory.
6. Next move to **Library Directories** and add our `C:\SDL2\lib\` folder. Now navigate to **Linker** | **Input** | **Additional Dependencies** and add `SDL2.lib`.
7. Click on **OK** and we are almost ready to build. We are now using `SDL2.lib`, so we can remove the `SDL.lib` and the `SDLmain.lib` files from the `SDL_image` project. Locate the files in the solution explorer, right-click and then remove the files. Change the build configuration to **release** and then build.
8. An error about being unable to start the program may appear. Just click on **OK** and we can close the project and continue.
9. There will now be a `Release` folder inside our `C:\SDL2_image\VisualC\` folder. Open it and copy the `SDL_image.dll` to our game's executable folder.
10. Next copy the `SDL2_image.lib` file into our original `C:\SDL2\lib\` directory. Also copy the `SDL_image` header from `C:\SDL2_image\` to the `C:\SDL2\include\` directory.
11. We just have a few more libraries to get and we are done. Download the `SDL_image-1.2.12-win32.zip` file (or the x64 if you are targeting a 64 bit platform) from `http://www.libsdl.org/projects/SDL_image/`. Extract all and then copy all of the `.dll` files apart from `SDL_image.dll` into our game's executable folder.

12. Open up our game project and go into its properties. Navigate to **Linker | Input | Additional Dependencies** and add `SDL2_image.lib`.



13. We have now installed `SDL_image` and can start to load all kinds of different image files. Copy the `animate.png` and `animate-alpha.png` images from the source downloads to our games assets folder and we can start loading PNG files.

Using SDL_image

So we have the library installed, now how do we use it? It is simple to use SDL_image in place of the regular SDL image loading. In our case we only need to replace one function and also add `#include <SDL_image.h>`.

```
SDL_Surface* pTempSurface = SDL_LoadBMP("assets/animate.bmp");
```

The preceding code will be changed as follows:

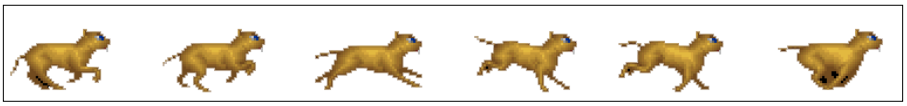
```
SDL_Surface* pTempSurface = IMG_Load("assets/animate.png");
```

We are now loading a `.png` image. PNG files are great to work with, they have a small file size and support an alpha channel. Let's perform a test. Change our renderer clear color to red.

```
SDL_SetRenderDrawColor(m_pRenderer, 255,0,0,255);
```

You will see that we still have our black background from the image we are using; this is definitely not ideal for our purposes.

When using PNG files, we can resolve this by using an alpha channel. We remove the background from the image and then when we load it, SDL will not draw anything from the alpha channel.



Let's load this image and see how it looks:

```
SDL_Surface* pTempSurface = IMG_Load("assets/animate-alpha.png");
```

This is exactly what we want:

Tying it into the framework

We have covered a lot on the subject of drawing images with SDL but we have yet to tie everything together into our framework so that it becomes reusable throughout our game. What we will now cover is creating a texture manager class that will have all of the functions we need to easily load and draw textures.

Creating the texture manager

The texture manager will have functions that allow us to load and create an `SDL_Texture` structure from an image file, draw the texture (either static or animated), and also hold a list of `SDL_Texture*`, so that we can use them whenever we need to. Let's go ahead and create the `TextureManager.h` file:

1. First we declare our `load` function. As parameters, the function takes the filename of the image we want to use, the ID we want to use to refer to the texture, and the renderer we want to use.

   ```
   bool load(std::string fileName,std::string id,
   SDL_Renderer* pRenderer);
   ```

2. We will create two draw functions, `draw` and `drawFrame`. They will both take the ID of the texture we want to draw, the `x` and `y` position we want to draw to, the height and width of the frame or the image we are using, the renderer we will copy to, and an `SDL_RendererFlip` value to describe how we want the image to be displayed (default is `SDL_FLIP_NONE`). The `drawFrame` function will take two additional parameters, the current frame we want to draw and which row it is on in the sprite sheet.

   ```
   // draw
   void draw(std::string id, int x, int y, int width, int
   height, SDL_Renderer* pRenderer, SDL_RendererFlip flip =
   SDL_FLIP_NONE);

   // drawframe

   void drawFrame(std::string id, int x, int y, int width, int
   height, int currentRow, int currentFrame, SDL_Renderer*
   pRenderer, SDL_RendererFlip flip = SDL_FLIP_NONE);
   ```

3. The `TextureManager` class will also contain `std::map` of pointers to the `SDL_Texture` objects, keyed using `std::strings`.

   ```
   std::map<std::string, SDL_Texture*> m_textureMap;
   ```

4. We now must define these functions in a `TextureManager.cpp` file. Let's start with the `load` function. We will take the code from our previous texture loading and use it within this `load` method.

```
bool TextureManager::load(std::string fileName, std::string
id, SDL_Renderer* pRenderer)
{
  SDL_Surface* pTempSurface = IMG_Load(fileName.c_str());

  if(pTempSurface == 0)
  {
    return false;
  }

  SDL_Texture* pTexture =
  SDL_CreateTextureFromSurface(pRenderer, pTempSurface);

  SDL_FreeSurface(pTempSurface);

  // everything went ok, add the texture to our list
  if(pTexture != 0)
  {
    m_textureMap[id] = pTexture;
    return true;
  }

  // reaching here means something went wrong
  return false;
}
```

5. When we call this function we will then have `SDL_Texture` that can be used by accessing it from the map using its ID; we will use this in our `draw` functions. The `draw` function can be defined as follows:

```
void TextureManager::draw(std::string id, int x, int y, int
width, int height, SDL_Renderer* pRenderer,
SDL_RendererFlip flip)
{
  SDL_Rect srcRect;
  SDL_Rect destRect;

  srcRect.x = 0;
  srcRect.y = 0;
  srcRect.w = destRect.w = width;
```

```
  srcRect.h = destRect.h = height;
  destRect.x = x;
  destRect.y = y;

  SDL_RenderCopyEx(pRenderer, m_textureMap[id], &srcRect,
  &destRect, 0, 0, flip);
}
```

6. We again use `SDL_RenderCopyEx` using the passed in ID variable to get the `SDL_Texture` object we want to draw. We also build our source and destination variables using the passed in `x`, `y`, `width`, and `height` values. Now we can move onto `drawFrame`:

```
void TextureManager::drawFrame(std::string id, int x, int y, int
width, int height, int currentRow, int currentFrame, SDL_Renderer
*pRenderer, SDL_RendererFlip flip)
{
  SDL_Rect srcRect;
  SDL_Rect destRect;
  srcRect.x = width * currentFrame;
  srcRect.y = height * (currentRow - 1);
  srcRect.w = destRect.w = width;
  srcRect.h = destRect.h = height;
  destRect.x = x;
  destRect.y = y;

  SDL_RenderCopyEx(pRenderer, m_textureMap[id], &srcRect,
  &destRect, 0, 0, flip);
}
```

   In this function, we create a source rectangle to use the appropriate frame of the animation using the `currentFrame` and `currentRow` variables. The source rectangle's `x` position for the current frame is the width of the source rectangle multiplied by the `currentFrame` value (covered in the *Animating a sprite sheet* section). Its `y` value is the height of the rectangle multiplied by `currentRow - 1` (it sounds more natural to use the first row, rather than the zeroth row).

7. We now have everything we need to easily load and draw textures throughout our game. Let's go ahead and test it out using the `animated.png` image. Open up `Game.h`. We will not need our texture member variables or the rectangles anymore, so delete any of the code dealing with them from the `Game.h` and `Game.cpp` files. We will however create two new member variables.

```
int m_currentFrame;
TextureManager m_textureManager;
```

8. We will use the `m_currentFrame` variable to allow us to animate our sprite sheet and we also need an instance of our new `TextureManager` class (ensure you include `TextureManager.h`). We can now load a texture in the game's `init` function.

```
m_textureManager.load("assets/animate-alpha.png",
"animate", m_pRenderer);
```

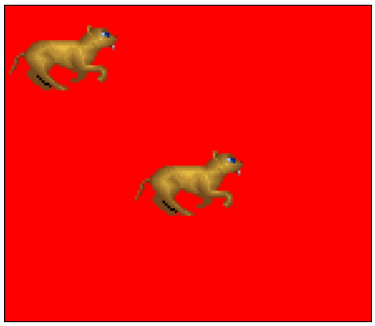
9. We have given this texture an ID of `"animate"` which we can use in our `draw` functions. We will start by drawing a static image at 0,0 and an animated image at 100,100. Here is the render function:

```
void Game::render()
{

  SDL_RenderClear(m_pRenderer);

  m_textureManager.draw("animate", 0,0, 128, 82,
  m_pRenderer);

  m_textureManager.drawFrame("animate", 100,100, 128, 82,
  1, m_currentFrame, m_pRenderer);

  SDL_RenderPresent(m_pRenderer);

}
```

10. The drawFrame function uses our `m_currentFrame` member variable. We can increment this in the `update` function like we did before, but we now do the calculation of the source rectangle inside the `draw` function.

```
void Game::update()
{
  m_currentFrame = int(((SDL_GetTicks() / 100) % 6));
}
```

Now we can build and see our hard work in action.



Using texture manager as a singleton

Now that we have our texture manager in place we still have one problem. We want to reuse this `TextureManager` throughout our game so we don't want it to be a member of our `Game` class because then we would have to pass it into our draw function. A good option for us is to implement `TextureManager` as a singleton. A singleton is a class that can only have one instance. This works for us, as we want to reuse the same `TextureManager` throughout our game. We can make our `TextureManager` a singleton by first making its constructor private.

```
private:

TextureManager() {}
```

This is to ensure that it cannot be created like other objects. It can only be created and accessed using the `Instance` function, which we will declare and define.

```
static TextureManager* Instance()
{
  if(s_pInstance == 0)
  {
    s_pInstance = new TextureManager();
    return s_pInstance;
  }

  return s_pInstance;
}
```

This function checks whether we already have an instance of our `TextureManager`. If not, then it constructs it, otherwise it simply returns the static instance. We will also `typedef` the `TextureManager`.

```
typedef TextureManager TheTextureManager;
```

We must also define the static instance in `TextureManager.cpp`.

```
TextureManager* TextureManager::s_pInstance = 0;
```

We can now use our `TextureManager` as a singleton. We no longer have to have an instance of `TextureManager` in our `Game` class, we just include the header and use it as follows:

```
// to load
if(!TheTextureManager::Instance()->load("assets/animate-alpha.png",
"animate", m_pRenderer))
{
   return false;
}
// to draw
TheTextureManager::Instance()->draw("animate", 0,0, 128, 82,
m_pRenderer);
```

When we load a texture in our `Game` (or any other) class we can then access it throughout our code.

Summary

This chapter has been all about rendering images onto the screen. We have covered source and destination rectangles and animating a sprite sheet. We took what we learned and applied it to creating a reusable texture manager class, enabling us to easily load and draw images throughout our game. In the next chapter, we will cover using inheritance and polymorphism to create a base game object class and use it within our game framework.